Assignment 7: Constructing Neural Networks

Overview

This assignment has goals around the following:

1) Comfort reading provided code and documentation.
2) Transforming a dataset into the right form as input to a supervised ML system.
3) Hands-on experience constructing Neural Networks.
4) Experience tuning a Neural Network to improve performance.

Neural nets are an incredibly complicated concept with many varieties. For this assignment we'll focus on the most standard of these implementations, a multilayer network with feed forward nodes. We want to give you a chance to tinker with and get a sense of how neural networks work. We are NOT asking you to create a neural network from scratch.

I also want to give you some experience with looking at and utilizing code written by other people. Additionally, you won't be writing code to tests, rather simply constructing some neural networks and attempting to reduce the network error (on the training data).

In this exercise, you will explore the basics of Neural Networks. If you'd like some more detail here are a few different resources for later:

- a video - https://www.youtube.com/watch?v=aircAruvnKk ,
- an article - https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6 (which actually references the video),
- and a longer form online text http://neuralnetworksanddeeplearning.com/ if you have the time/interest.

What you'll turn in (upload to canvas)

This assignment will be very non-traditional in that you'll be exploring and writing some code to deal with data sets and construct Neural Networks. However, I don't want you to submit your code. Instead, I'd like you to keep (and turn in) "lab notes" on what you try. I recommend that you start a word doc or excel sheet and take notes on each task. The most important details are the configurations of each network that you create and the resulting "error" values. These notes don't need to be formal at all. Here is an example excerpt from my lab notes:

Task 3: Wine dataset
13 inputs, 1 output
all data was normalized too 0 to 1 range
sigmoid activation function and 3 hidden nodes
learning rate was 0.5
Error after 10000 iterations: 0.003731905985943266

Task 1:
In the provided zip, you'll see a main.py. I recommend writing code to complete the remaining tasks in that main.py, perhaps with comments to denote which task you are on.

Try the following instructions to construct a neural network to compute the xor function.

First, you'll need to import the neural library so that we can use it. For convenience (so that we don't have to type neural.NeuralNet), we'll use the "from … import" syntax

```
from neural import NeuralNet
```

To use the provided NeuralNet class, we must first construct some training data.

The training data is a list of input-output tuples (pairs). Each item in this tuple - the input and output separately are specified as lists. In this case, our input is a list of two elements and our output is a list of one element.

```
# each row is an (input, output) tuple
xor_data = [
    #   input     output    corresponding example
    ([0.0, 0.0],  [0.0]),  #[0, 0] => 0
    ([0.0, 1.0],  [1.0]),  #[0, 1] => 1
    ([1.0, 0.0],  [1.0]),  #[1, 1] => 1
    ([1.0, 1.0],  [0.0])   #[1, 0] => 0
]
```

Important note: We are going to train AND test this neural network on the same xor data. This is clearly not a good idea (for the reasons discussed in previous assignments). However, for xor, there are only 4 possible examples, so separate training and testing sets are not possible. For this assignment more generally, the focus is not on evaluation (as it was for our k-fold cross validation assignment). Instead, the focus is on Neural Networks and experimentation. As we train our networks, we'll look at (and record) the error over the training data. The error is calculated as the sum of squared errors over the examples.

Now we can create and train an instance of the NeuralNet class. Here is an example of creating and training a network using the class we've provided. The Neural Network has two input nodes, five hidden nodes, and one output node. We're only creating Neural Networks with one hidden layer, though feel free to have as many nodes in that layer as you'd like.

```
nn = NeuralNet(2, 5, 1)
nn.train(xor_data)
```

When I run the above code, I get the following output:

```
Error after 100 iterations: 0.47427457513158167
```

```
Error after 200 iterations: 0.3716547002426295
Error after 300 iterations: 0.21978817009149293
Error after 400 iterations: 0.09270142732780062
Error after 500 iterations: 0.04520901028203348
Error after 600 iterations: 0.02712849580670401
Error after 700 iterations: 0.018577053921847882
Error after 800 iterations: 0.013817682578923771
Error after 900 iterations: 0.010857930403733252
Error after 1000 iterations: 0.008868060166711978
```

These messages indicate that we have trained our Neural Network for 1000 iterations/epochs. In each epoch, all 4 examples were passed through the network one at a time. Weights are updated after each example, to improve the system. After these 1000 epochs, the sum of squared errors (over the training data set) is 0.00887, which is very small. However, this number is not very meaningful to us. The functions that follow enable to us to dig a little bit deeper to investigate the network performance.

Once training has finished there are several methods to use/test our network. The three methods are *evaluate*, *test*, and *test_with_expected*. Here's a little more detail on each (for a lengthier description see "The Neural Network Class" description in documentation.pdf):

- evaluate(self, inputs: List[Any]) -> List[float] - takes a single input (which remember is a list itself) and returns the output as a list (even if the output is only one element). The two other methods in this list each use *evaluate* in their implementation.
- test(self, data: List[List[Any]]) -> List[Tuple[List[Any], List[Any]]] - takes a list of input examples and returns a list of (input, output) tuples
- test_with_expected(self, data: List[Tuple[List[Any], List[Any]]]) -> List[Tuple[List[Any], List[Any], List[Any]]]: - takes a list of (input, expected_output) tuples and returns a list of (input, expected_output, actual_output) triples. It might seem weird to pass in expected outputs but this sort of method can be helpful when trying to evaluate the accuracy of your model to compare expected and actual outputs.

To run a single example through our network….

```
nn.evaluate([0.0, 1.0])
```

The above would return [1.0] if the network were a "perfect" xor. It won't be perfect, so should be close to 1.0, but not exactly 1.0. For example, when I run the above code, I get the following output.

```
[0.9392848146509899]
```

The output of either testing function can be difficult to read. You can make it easier to read if you print each result on its own line:

```
for triple in nn.test_with_expected(xor_data):
    print(triple)
```

For example, when I run the above code, I get the following output

```
([0.0, 0.0], [0.0], [0.05946921608477217])
([0.0, 1.0], [1.0], [0.9392848146509899])
([1.0, 0.0], [1.0], [0.9244695364307255])
([1.0, 1.0], [0.0], [0.0680579981156151])
```

Each line gives the inputs, expected output, and actual output (generated by the trained network).

Finally, looking at the input and outputs (expected and actual) for each example is useful, but if we have many inputs, it becomes more difficult to view. In that case, it might better to simply look at the expected output and actual output only.

```
for i in nn.test_with_expected(xor_data):
    print(f"desired: {i[1]}, actual: {i[2]}")
```

When I run the above code, I get the following output:

```
desired: [0.0], actual: [0.05946921608477217]
desired: [1.0], actual: [0.9392848146509899]
desired: [1.0], actual: [0.9244695364307255]
desired: [0.0], actual: [0.0680579981156151]
```

In the text above, I've just been introducing you to the provided neural net library. Now, using it, here are some other things to explore…..

- Notice the optional (keyword) parameters in the train method. Try altering the print_interval (how often the error is printed), number of iterations (epochs), and learning rate.
- Change the number of hidden nodes. How does the error change?
- Not that you'd want to (because they don't really carry "human readable" meaning), but if you needed to look at the learned weights, how could you view them?

Task 2:

Try the same task above, but now only with 1 hidden node instead of 5. Using 1 hidden node simulates a perceptron. From our theoretical discussion of perceptrons, we know that a perceptron cannot be trained (well) for the xor function. To observe that the system isn't trained well, examine the error after 1000 iterations. Additionally inspect the desired/expected and actual outputs for the examples.

Task 3:

While the xor function is important in the history of neural networks, let's move on to some more interesting datasets. Let's try the "wine" dataset.
https://archive.ics.uci.edu/ml/datasets/wine

Notice that wine.data and wine.names is already included in your project folder (inside “data”).

At a high level, the data contains chemical analysis of wines from 3 different wineries. Each line, corresponding to a wine bottle, starts with a 1, 2 or 3 indicating which winery it came from. The winery source is the target value, the value that we want to predict.

We want to construct a neural net, using the remaining 13 attributes on each line to predict the source winery, the first value (1, 2 or 3). While the data is all numerical and nicely structured, the values are in varying ranges. To effectively construct and train a neural network, we need to load the data from the csv file into python. Next, we need to normalize this data, so that the values for each attribute range from 0 to 1. Similarly, we should normalize the output values between 0 and 1. For example, after normalizing the output, 1 becomes 0.0, 2 becomes 0.5 and 3 becomes 1.0.

After getting your data into a workable format, construct and train a neural network. In addition to training error, I recommend that you print the desired and actual outputs so that you can investigate the network performance.

Notice that I included my lab notes in sara_wine_results.xlsx so that you can see baseline performance for comparison.

Task 4 (optional):
While the wine data set was tackling a more realistic problem, it was idealistic in that all the attributes were continuous. We had to normalize the data, however, other data sets often contain categorical attributes that require more attention.

Let’s look at an absenteeism data set:
https://archive.ics.uci.edu/ml/datasets/Absenteeism+at+work
Notice the related files stored in your project folder.

The task in this case is to predict the target value, “Absenteeism time in hours”, which is the last value on each line. While we don’t have a lot of description about this dataset, let’s assume that each line represents an employee not showing up at work. We want to predict how long this absence will last (in hours). Again, as in the wine task, we need to normalize this data so that the attribute values range from 0 to 1.

In this data set, there are 3 attributes that need special attention beyond simple normalization:

1) The first attribute, “id”, should not be used in training at all as an individual’s identity won’t help once we move out of the training context.
2) The second attribute, “reason for absence”, might include very useful information, but simply normalizing a categorical attribute won’t produce useful values. There are 29 possible values for this attribute, described in the data documentation. Instead of normalizing this attribute, I recommend that you translate this one categorical attribute into 29 binary attributes, each corresponding to one of the possible values.

3) Similarly, the fifth attribute, “seasons”, is a categorical attribute with four possible values – 1 (meaning summer), 2 (meaning autumn), 3 (meaning winter), or 4 (meaning spring). I suggest translating this one categorical attribute into 4 binary attributes. For example, if you have an example with a value 2, that would translate to [0.0, 1.0, 0.0, 0.0].

After getting your data into a workable format, construct and train a neural network.

Extensions
Explore other datasets here:
https://archive.ics.uci.edu/ml/datasets.php

Or check out some Kaggle competitions!